TODAY'S AGENDA

Background
Parallel Hash Join
Hash Functions
Hashing Schemes
Evaluation
PARALLEL JOIN ALGORITHMS

Perform a join between two relations on multiple threads simultaneously to speed up operation.

Two main approaches:
→ Hash Join
→ Sort-Merge Join

We won’t discuss nested-loop joins…
Many OLTP DBMSs don’t implement hash join.

But an **index nested-loop join** with a small number of target tuples is more or less equivalent to a hash join.
HASHING VS. SORTING

1970s – Sorting
1980s – Hashing
1990s – Equivalent
2000s – Hashing
2010s – Hashing (Partitioned vs. Non-Partitioned)
2020s – ???
# Parallel Join Algorithms

<table>
<thead>
<tr>
<th><strong>Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>VLDB 2009</td>
</tr>
<tr>
<td>→ Hashing is faster than Sort-Merge.</td>
</tr>
<tr>
<td>→ Sort-Merge is faster w/ wider SIMD.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>VLDB 2012</td>
</tr>
<tr>
<td>→ Sort-Merge is already faster than Hashing, even without SIMD.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ICDE 2013</td>
</tr>
<tr>
<td>→ New optimizations and results for Radix Hash Join.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGMOD 2011</td>
</tr>
<tr>
<td>→ Trade-offs between partitioning &amp; non-partitioning Hash-Join.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Massively Parallel NUMA-Aware Hash Joins</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>IMDM 2013</td>
</tr>
<tr>
<td>→ Ignore what we said last year.</td>
</tr>
<tr>
<td>→ You really want to use Hashing!</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGMOD 2016</td>
</tr>
<tr>
<td>→ Hold up everyone! Let's look at everything for real!</td>
</tr>
</tbody>
</table>
JOIN ALGORITHM DESIGN GOALS

Goal #1: Minimize Synchronization
→ Avoid taking latches during execution.

Goal #2: Minimize CPU Cache Misses
→ Ensure that data is always local to worker thread.
IMPROVING CACHE BEHAVIOR

Factors that affect cache misses in a DBMS:
→ Cache + TLB capacity.
→ Locality (temporal and spatial).

Non-Random Access (Scan):
→ Clustering to a cache line.
→ Execute more operations per cache line.

Random Access (Lookups):
→ Partition data to fit in cache + TLB.

Source: Johannes Gehrke
PARALLEL HASH JOINS

Hash join is the most important operator in a DBMS for OLAP workloads.

It’s important that we speed it up by taking advantage of multiple cores.
→ We want to keep all of the cores busy, without becoming memory bound
HASH JOIN (R⨝S)

Phase #1: Partition (optional)
→ Divide the tuples of R and S into sets using a hash on the join key.

Phase #2: Build
→ Scan relation R and create a hash table on join key.

Phase #3: Probe
→ For each tuple in S, look up its join key in hash table for R. If a match is found, output combined tuple.
PARTITION PHASE

Split the input relations into partitioned buffers by hashing the tuples’ join key(s).

→ Ideally the cost of partitioning is less than the cost of cache misses during build phase.

→ Sometimes called hybrid hash join.

Contents of buffers depends on storage model:

→ **NSM**: Either the entire tuple or a subset of attributes.

→ **DSM**: Only the columns needed for the join + offset.
PARTITION PHASE

Approach #1: Non-Blocking Partitioning
→ Only scan the input relation once.
→ Produce output incrementally.

Approach #2: Blocking Partitioning (Radix)
→ Scan the input relation multiple times.
→ Only materialize results all at once.
→ Sometimes called *radix hash join*.
NON-BLOCKING PARTITIONING

Scan the input relation only once and generate the output on-the-fly.

Approach #1: Shared Partitions
→ Single global set of partitions that all threads update.
→ Have to use a latch to synchronize threads.

Approach #2: Private Partitions
→ Each thread has its own set of partitions.
→ Have to consolidate them after all threads finish.
### Data Table

The table is partitioned based on the hash function $\text{hash}_p(key)$. The hash values are distributed across different partitions (A, B, C) as follows:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

**SHARED PARTITIONS**

The sharing of partitions helps in distributing the load evenly across different nodes in a distributed system.
**SHARED PARTITIONS**

**Data Table**

- Columns: A, B, C
- Hash function: $hash_p(key)$

**Partitions**

- $P_1$, $P_2$, ..., $P_n$
- Each partition contains records

Note: The diagram illustrates how the data table is partitioned and distributed among the partitions.
**SHARED PARTITIONS**

Data Table

\[ \text{hash}_p(key) \]

Partitions

\[ P_1 \]
\[ P_2 \]
\[ \vdots \]
\[ P_n \]
**SHARE PARTITIONS**

**Data Table**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>hash_p(key)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_n</td>
</tr>
</tbody>
</table>

**Partitions**

P_1, P_2, ..., P_n
**SHARED PARTITIONS**

### Data Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Partitions

$$\text{hash}_p(key)$$

- $P_1$
- $P_2$
- $P_n$

[Diagram showing data table and partitions connected by hash function]$\text{hash}_p(key)$
PRIVATE PARTITIONS

Data Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>hash(p(key))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(#_p)</td>
</tr>
</tbody>
</table>

Partitions

\(P_1\)  \(P_2\)  \(P_n\)

\(P_1\)  \(P_2\)  \(P_n\)

\(P_1\)  \(P_2\)  \(P_n\)
PRIVATE PARTITIONS

Data Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>hash_p(key)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partitions

\[ P_1, P_2, \ldots, P_n \]

Combined

\[ P_1, P_2, \ldots, P_n \]
# PRIVATE PARTITIONS

## Data Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="key" alt="hash_p" /></td>
<td><img src="key" alt="hash_p" /></td>
<td><img src="key" alt="hash_p" /></td>
</tr>
</tbody>
</table>

## Partitions

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="key" alt="hash_p" /></td>
<td><img src="key" alt="hash_p" /></td>
<td><img src="key" alt="hash_p" /></td>
</tr>
</tbody>
</table>

## Combined

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="key" alt="hash_p" /></td>
<td><img src="key" alt="hash_p" /></td>
<td><img src="key" alt="hash_p" /></td>
</tr>
</tbody>
</table>
PRIVATE PARTITIONS

Data Table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>hash_p(key)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partitions

\[ P_1, P_2, \ldots, P_n \]

Combined

\[ P_1, P_2, \ldots, P_n \]
**RADIX PARTITIONING**

Scan the input relation multiple times to generate the partitions.

Multi-step pass over the relation:
- **Step #1**: Scan $R$ and compute a histogram of the # of tuples per hash key for the **radix** at some offset.
- **Step #2**: Use this histogram to determine output offsets by computing the **prefix sum**.
- **Step #3**: Scan $R$ again and partition them according to the hash key.
The radix is the value of an integer at a particular position (using its base).

**Input**

89 12 23 08 41 64
The radix is the value of an integer at a particular position (using its base).

Input: 891223084164
Radix: 923814
The radix is the value of an integer at a particular position (using its base).
The prefix sum of a sequence of numbers 
\((x_0, x_1, ..., x_n)\)
is a second sequence of numbers 
\((y_0, y_1, ..., y_n)\)
that is a running total of the input sequence.

### Input

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

**PREFIX SUM**
The prefix sum of a sequence of numbers 
\((x_0, x_1, ..., x_n)\)
is a second sequence of numbers 
\((y_0, y_1, ..., y_n)\)
that is a running total of the input sequence.

**Input**

| 1 | 2 | 3 | 4 | 5 | 6 |

**Prefix Sum**

| 1 |
The prefix sum of a sequence of numbers 
\((x_0, x_1, ..., x_n)\)
is a second sequence of numbers 
\((y_0, y_1, ..., y_n)\)
that is a running total of the input sequence.
The prefix sum of a sequence of numbers $(x_0, x_1, ..., x_n)$ is a second sequence of numbers $(y_0, y_1, ..., y_n)$ that is a running total of the input sequence.
**RADIX PARTITIONS**

*Step #1: Inspect input, create histograms*

<table>
<thead>
<tr>
<th>#_p</th>
<th>07</th>
</tr>
</thead>
<tbody>
<tr>
<td>#_p</td>
<td>18</td>
</tr>
<tr>
<td>#_p</td>
<td>19</td>
</tr>
<tr>
<td>#_p</td>
<td>07</td>
</tr>
<tr>
<td>#_p</td>
<td>03</td>
</tr>
<tr>
<td>#_p</td>
<td>11</td>
</tr>
<tr>
<td>#_p</td>
<td>15</td>
</tr>
<tr>
<td>#_p</td>
<td>10</td>
</tr>
</tbody>
</table>
RADIX PARTITIONS

Step #1: Inspect input, create histograms

<table>
<thead>
<tr>
<th>hash_p(key)</th>
<th>07</th>
<th>18</th>
<th>19</th>
<th>07</th>
<th>03</th>
<th>11</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>#p</td>
<td>#p</td>
<td>#p</td>
<td>#p</td>
<td>#p</td>
<td>#p</td>
<td>#p</td>
<td>#p</td>
<td>#p</td>
</tr>
</tbody>
</table>
# Radix Partitions

**Step #1: Inspect input, create histograms**

<table>
<thead>
<tr>
<th>hash_p(key)</th>
<th>0 7</th>
<th>1 8</th>
<th>1 9</th>
<th>0 7</th>
<th>0 3</th>
<th>1 1</th>
<th>1 5</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#_p</td>
<td>#_p</td>
<td>#_p</td>
<td>#_p</td>
<td>#_p</td>
<td>#_p</td>
<td>#_p</td>
<td>#_p</td>
<td>#_p</td>
</tr>
</tbody>
</table>

\[
\text{hash}_p(key)
\]
RADIX PARTITIONS

Step #1: Inspect input, create histograms

<table>
<thead>
<tr>
<th>hash_p(key)</th>
<th>07</th>
<th>18</th>
<th>19</th>
<th>07</th>
<th>03</th>
<th>11</th>
<th>15</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>#p 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#p 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partition 0: 2
Partition 1: 2

Partition 0: 1
Partition 1: 3
RADIX PARTITIONS

Step #2: Compute output offsets

Partition 0:
- CPU 1
- 2

Partition 1:
- CPU 1
- 2

Partition 0:
- CPU 0
- 1

Partition 1:
- CPU 0
- 3
RADIX PARTITIONS

Step #3: Read input and partition

Partition 0, CPU 0
Partition 0, CPU 1
Partition 1, CPU 0
Partition 1, CPU 1
RADI X PARTITIONS

Step #3: Read input and partition

Partition 0: 2
Partition 1: 2

Partition 0: 1
Partition 1: 3

Partition 0, CPU 0
Partition 0, CPU 1
Partition 1, CPU 0
Partition 1, CPU 1
RADIX PARTITIONS

Recursively repeat until target number of partitions have been created

Partition 0:
- 07
- 18
- 19
- 07
- 03
- 11
- 15
- 10

Partition 0: 2
Partition 1: 2

Partition 1:
- 07
- 07
- 03
- 18
- 19
- 11
- 15
- 10

Partition 0: 1
Partition 1: 3
## RADIX PARTITIONS

Recursively repeat until target number of partitions have been created

| hash_p(key) | #p | | #p | | #p | | #p | | #p | |
|-------------|----|---|----|---|----|---|----|---|----|
|             | 07 | 18| 19 | 07 | 03 | 11 | 15 | 10 |     |
| Partition 0: |   |   |    | 2 |    |    |    |    | 1   |
| Partition 1: |   |   |    | 2 |    |    |    |    | 3   |
|             | 07 | 07| 03 | 18 | 19 | 11 | 15 | 10 |     |
| Partition 0: |   |   |    |   |    |    |    |    | 1   |
| Partition 1: |   |   |    |   |    |    |    |    | 3   |
### RADIX PARTITIONS

Recursively repeat until target number of partitions have been created

<table>
<thead>
<tr>
<th>hash(key)</th>
<th>#p</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>07</td>
<td>07</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>07</td>
<td>07</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>03</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

**Partition 0:** 2
**Partition 1:** 2

**Partition 0:** 1
**Partition 1:** 3
BUILD PHASE

The threads are then to scan either the tuples (or partitions) of \( R \).

For each tuple, hash the join key attribute for that tuple and add it to the appropriate bucket in the hash table.
→ The buckets should only be a few cache lines in size.
Design Decision #1: Hash Function
→ How to map a large key space into a smaller domain.
→ Trade-off between being fast vs. collision rate.

Design Decision #2: Hashing Scheme
→ How to handle key collisions after hashing.
→ Trade-off between allocating a large hash table vs. additional instructions to find/insert keys.
We don't want to use a cryptographic hash function for our join algorithm.

We want something that is fast and will have a low collision rate.

→ **Best Speed**: Always return '1'

→ **Best Collision Rate**: Perfect hashing
HASH FUNCTIONS

**CRC-32** **(1975)**
→ Used in networking for error detection.

**MurmurHash** **(2008)**
→ Designed to a fast, general purpose hash function.

**Google CityHash** **(2011)**
→ Designed to be faster for short keys (<64 bytes).

**Google FarmHash** **(2014)**
→ Newer version of CityHash with better collision rates.

**CLHash** **(2016)**
→ Fast hashing function based on carry-less multiplication.
**HASH FUNCTION BENCHMARK**

*Intel Core i7-8700K @ 3.70GHz*

- crc32
- std::hash
- MurmurHash3
- CityHash
- FarmHash
- CLHash

Throughput (MB/sec) vs. Key Size (bytes)

Source: Fredrik Widlund
HASH FUNCTION BENCHMARK

Intel Core i7-8700K @ 3.70GHz

Throughput (MB/sec)

Key Size (bytes)

Source: Fredrik Widlund
HASHING SCHEMES

Approach #1: Chained Hashing
Approach #2: Linear Probe Hashing
Approach #3: Robin Hood Hashing
Approach #4: Cuckoo Hashing
CHAINED HASHING

Maintain a linked list of buckets for each slot in the hash table.

Resolve collisions by placing all elements with the same hash key into the same bucket.

→ To determine whether an element is present, hash to its bucket and scan for it.

→ Insertions and deletions are generalizations of lookups.
CHAINED HASHING

\[
\text{hash(key)} \rightarrow \text{Buckets}
\]
LINEAR PROBE HASHING

Single giant table of slots.

Resolve collisions by linearly searching for the next free slot in the table.
→ To determine whether an element is present, hash to a location in the index and scan for it.
→ Have to store the key in the index to know when to stop scanning.
→ Insertions and deletions are generalizations of lookups.
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A
LINEAR PROBE HASHING

\text{hash(key)}

\begin{align*}
\text{hash(A)} & \mid A \\
\text{hash(B)} & \mid B \\
\end{align*}
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A

hash(B) | B

hash(C) | C
LINEAR PROBE HASHING

hash(key)

\[
\begin{align*}
\text{hash}(A) & : A \\
\text{hash}(B) & : B \\
\text{hash}(A) & : A \\
\text{hash}(C) & : C \\
\text{hash}(D) & : D
\end{align*}
\]
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

hash(A) | A
hash(B) | B
hash(C) | C
hash(D) | D
hash(E) | E
LINEAR PROBE HASHING

\[ \text{hash(key)} \]

\[
\begin{array}{c}
\text{hash(A)} | A \\
\text{hash(B)} | B \\
\text{hash(C)} | C \\
\text{hash(D)} | D \\
\text{hash(E)} | E \\
\text{hash(F)} | F \\
\end{array}
\]
OBSERVATION

To reduce the # of wasteful comparisons during the join, it is important to avoid collisions of hashed keys.

This requires a chained hash table with ~2x the number of slots as the # of elements in R.
ROBIN HOOD HASHING

Variant of linear hashing that steals slots from "rich" keys and give them to "poor" keys.
→ Each key tracks the number of positions they are from where its optimal position in the table.
→ On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.
ROBIN HOOD HASHING

- hash(key)
  - hash(A) \[ A[0] \]
  - # of "Jumps" From First Position
ROBIN HOOD HASHING

hash(key)

hash(A) | A [\emptyset]

hash(B) | B [\emptyset]
ROBIN HOOD HASHING

\[ \text{hash}(\text{key}) \]

A
B
C
D
E
F

hash(B) | B [\emptyset]

hash(A) | A [\emptyset]

A[0] == C[0]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B [0]

hash(A) | A [0]

hash(C) | C [1]

A[0] == C[0]
ROBIN HOOD HASHING

\[ \text{hash(key)} \]

\[
\begin{array}{c|c}
A & B[\emptyset] \\
B & A[\emptyset] \\
C & C[1] \\
D & \\
E & \\
F & \\
\end{array}
\]

C[1] > D[0]
ROBIN HOOD HASHING

$\text{hash}(\text{key})$

- $A$
- $B$
- $C$
- $D$
- $E$
- $F$

$\text{hash}(A) | A [\emptyset]$
$\text{hash}(B) | B [\emptyset]$
$\text{hash}(C) | C [1]$
$\text{hash}(D) | D [1]$

$C[1] > D[0]$
ROBIN HOOD HASHING

hash(key)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

hash(A) | A [0]
hash(B) | B [Ø]
hash(C) | C [1]
hash(D) | D [1]
ROBIN HOOD HASHING

<table>
<thead>
<tr>
<th>hash(key)</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(A)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash(B)</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash(C)</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash(D)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- A[0] == E[0]
- C[1] == E[1]
ROBIN HOOD HASHING

hash(key)

A  B  C  D  E  F

<table>
<thead>
<tr>
<th>hash(B)</th>
<th>B [∅]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hash(A)</th>
<th>A [∅]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hash(C)</th>
<th>C [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hash(E)</th>
<th>E [2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A[0] == E[0]
C[1] == E[1]
ROBIN HOOD HASHING

- **hash(key)**

- **A**
- **B**
- **C**
- **D**
- **E**
- **F**

- **hash(B)** | **B [0]**
- **hash(A)** | **A [0]**
- **hash(C)** | **C [1]**
- **hash(E)** | **E [2]**
- **hash(D)** | **D [2]**

- **A[0] == E[0]**
- **C[1] == E[1]**
- **D[1] < E[2]**
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

hash(B) | B [∅]

hash(A) | A [∅]

hash(C) | C [1]

hash(E) | E [2]

hash(D) | D [2]

hash(F) | F [1]

D[2] > F[0]
CUCKOO HASHING

Use multiple tables with different hash functions.
→ On insert, check every table and pick anyone that has a free slot.
→ If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

Look-ups are always O(1) because only one location per hash table is checked.
Cuckoo Hashing

Insert X

hash₁(X)  hash₂(X)

Hash Table #1

Hash Table #2
Cuckoo Hashing

Insert X

hash_1(X)  hash_2(X)
Cuckoo Hashing

Hash Table #1

Insert X
\( \text{hash}_1(X) \) \( \text{hash}_2(X) \)

Insert Y
\( \text{hash}_1(Y) \) \( \text{hash}_2(Y) \)

Hash Table #2
Cuckoo Hashing

Hash Table #1

Insert X
\[ \text{hash}_1(X) \quad \text{hash}_2(X) \]

Insert Y
\[ \text{hash}_1(Y) \quad \text{hash}_2(Y) \]

Hash Table #2

\[ \text{hash}_2(Y) \quad Y \]
CUCKOO HASHING

Hash Table #1

Insert X
\[ \text{hash}_1(X) \quad \text{hash}_2(X) \]

Insert Y
\[ \text{hash}_1(Y) \quad \text{hash}_2(Y) \]

Insert Z
\[ \text{hash}_1(Z) \quad \text{hash}_2(Z) \]

Hash Table #2

\[ \text{hash}_2(Y) \mid Y \]
Cuckoo Hashing

Hash Table #1

Insert X
\[ \text{hash}_1(X) \quad \text{hash}_2(X) \]

Insert Y
\[ \text{hash}_1(Y) \quad \text{hash}_2(Y) \]

Insert Z
\[ \text{hash}_1(Z) \quad \text{hash}_2(Z) \]

Hash Table #2

\[ \text{hash}_2(Z) \mid Z \]
CUCKOO HASHING

Hash Table #1

Insert X
$\text{hash}_1(X)$  $\text{hash}_2(X)$

Insert Y
$\text{hash}_1(Y)$  $\text{hash}_2(Y)$

Insert Z
$\text{hash}_1(Z)$  $\text{hash}_2(Z)$  $\text{hash}_1(Y)$

Hash Table #2

$\text{hash}_2(Z)$  $Z$
Cuckoo Hashing

**Hash Table #1**

- Insert X:
  - $hash_1(X)$
  - $hash_2(X)$

- Insert Y:
  - $hash_1(Y)$
  - $hash_2(Y)$

- Insert Z:
  - $hash_1(Z)$
  - $hash_2(Z)$
  - $hash_1(Y)$

**Hash Table #2**

- Insert Y:
  - $hash_2(Z)$

- Insert Z:
  - $hash_2(Z)$

- Insert X:
  - $hash_1(X)$

- Insert Y:
  - $hash_1(Y)$
**Cuckoo Hashing**

**Hash Table #1**

- Insert X
  - \( \text{hash}_1(X) \)  \( \text{hash}_2(X) \)
- Insert Y
  - \( \text{hash}_1(Y) \)  \( \text{hash}_2(Y) \)
- Insert Z
  - \( \text{hash}_1(Z) \)  \( \text{hash}_2(Z) \)  \( \text{hash}_1(Y) \)  \( \text{hash}_2(X) \)

**Hash Table #2**

- \( \text{hash}_2(Z) \)  \( Z \)
- \( \text{hash}_2(X) \)  \( X \)
Threads have to make sure that they don’t get stuck in an infinite loop when moving keys.

If we find a cycle, then we can rebuild the entire hash tables with new hash functions.

→ With two hash functions, we (probably) won’t need to rebuild the table until it is at about 50% full.

→ With three hash functions, we (probably) won’t need to rebuild the table until it is at about 90% full.
PROBE PHASE

For each tuple in $S$, hash its join key and check to see whether there is a match for each tuple in corresponding bucket in the hash table constructed for $R$.

→ If inputs were partitioned, then assign each thread a unique partition.
→ Otherwise, synchronize their access to the cursor on $S$
Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called *sideways information passing*.
Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.
→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.
→ Sometimes called *sideways information passing*. 

\[ \text{Bloom Filter} \]
**PROBE PHASE – BLOOM FILTER**

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called *sideways information passing*.
Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.  
→ Threads check the filter before probing the hash table.  
   This will be faster since the filter will fit in CPU caches.  
→ Sometimes called *sideways information passing*.
## HASH JOIN VARIANTS

<table>
<thead>
<tr>
<th></th>
<th>No-P</th>
<th>Shared-P</th>
<th>Private-P</th>
<th>Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Input scans</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Sync during partitioning</td>
<td>_</td>
<td>Spinlock per tuple</td>
<td>Barrier, once at end</td>
<td>Barrier, 4 · #passes</td>
</tr>
<tr>
<td>Hash table</td>
<td>Shared</td>
<td>Private</td>
<td>Private</td>
<td>Private</td>
</tr>
<tr>
<td>Sync during build phase</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Sync during probe phase</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
BENCHMARKS

Primary key – foreign key join
→ Outer Relation (Build): 16M tuples, 16 bytes each
→ Inner Relation (Probe): 256M tuples, 16 bytes each

Uniform and highly skewed (Zipf; s=1.25)

No output materialization
HASH JOIN – UNIFORM DATA SET

Intel Xeon CPU X5650 @ 2.66GHz
6 Cores with 2 Threads Per Core

- Partition
- Build
- Probe

Source: Spyros Blanas
HASH JOIN – SKEWED DATA SET

Intel Xeon CPU X5650 @ 2.66GHz
6 Cores with 2 Threads Per Core

<table>
<thead>
<tr>
<th></th>
<th>Partition</th>
<th>Build</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Partitioning</td>
<td>25.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Partitioning</td>
<td>167.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private Partitioning</td>
<td>56.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix</td>
<td>50.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: Spyros Blanas
OBSERVATION

We have ignored a lot of important parameters for all of these algorithms so far.
→ Whether to use partitioning or not?
→ How many partitions to use?
→ How many passes to take in partitioning phase?

In a real DBMS, the optimizer will select what it thinks are good values based on what it knows about the data (and maybe hardware).
RADIX HASH JOIN – UNIFORM DATA SET

Intel Xeon CPU X5650 @ 2.66GHz
Varying the # of Partitions

Cycles / Output Tuple

<table>
<thead>
<tr>
<th>Radix / 1-Pass</th>
<th>64</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
<th>8192</th>
<th>32768</th>
<th>131072</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Build</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Probe</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Radix / 2-Pass</th>
<th>64</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>4096</th>
<th>8192</th>
<th>32768</th>
<th>131072</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Build</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Probe</td>
<td>-5%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Source: Spyros Blanas

Intel Xeon CPU X5650 @ 2.66GHz
Varying the # of Partitions

- No Partitioning
RADIX HASH JOIN – UNIFORM DATA SET

Intel Xeon CPU X5650 @ 2.66GHz
Varying the # of Partitions

- Partition
- Build
- Probe

Cycles / Output Tuple

Source: Spyros Blanas
EFFECTS OF HYPER-THREADING

Intel Xeon CPU X5650 @ 2.66GHz
Uniform Data Set

Multi-threading hides cache & TLB miss latency.
Radix join has fewer cache & TLB misses but this has marginal benefit.
Non-partitioned join relies on multi-threading for high performance.
PARTING THOUGHTS

On modern CPUs, a simple hash join algorithm that does not partition inputs is competitive.

There are additional vectorization execution optimizations that are possible in hash joins that we didn’t talk about. But these don’t really help...
NEXT CLASS

Parallel Sort-Merge Joins